REASSURE: A Self-contained Mechanism for Healing Software Using Rescue Points

Georgios Portokalidis and Angelos D. Keromytis

Network Security Lab, Department of Computer Science,
Columbia University, New York, NY, USA
{porto,angelos}@cs.columbia.edu

Abstract. Software errors are frequently responsible for the limited availability of Internet Services, loss of data, and many security compromises. Self-healing using rescue points (RPs) is a mechanism that can be used to recover software from unforeseen errors until a more permanent remedy, like a patch or update, is available. We present REASSURE, a self-contained mechanism for recovering from such errors using RPs. Essentially, RPs are existing code locations that handle certain anticipated errors in the target application, usually by returning an error code. REASSURE enables the use of these locations to also handle unexpected faults. This is achieved by rolling back execution to a RP when a fault occurs, returning a valid error code, and enabling the application to gracefully handle the unexpected error itself. REASSURE can be applied on already running applications, while disabling and removing it is equally facile. We tested REASSURE with various applications, including the MySQL and Apache servers, and show that it allows them to successfully recover from errors, while incurring moderate overhead between 1% and 115%. We also show that even under very adverse conditions, like their continuous bombardment with errors, REASSURE protected applications remain operational.

1 Introduction

Program errors or bugs are ever-present in software, and specially in large and highly complex code bases [20]. They manifest as application crashes or unexpected behavior and can cause significant problems, like limited availability of Internet services [22], loss of user data [11], or lead to system compromise [24]. Many attempts have been made to increase the quality of software and reduce the number of bugs. Companies enforce strict development strategies and educate their developers in proper development practices, while static and dynamic analysis tools are used to assist in bug discovery [2,5]. However, it has been established that it is extremely difficult to produce completely error-free software.

To alleviate some of the dangers that bugs like buffer overflows and dangling pointers entail, various containment and runtime protection techniques have been proposed [8,1,7,12,18]. These techniques can offer assurances that certain types of program vulnerabilities cannot be exploited to compromise security,

but they do not also offer high availability and reliability, as they frequently terminate the compromised program to prevent the attacker from performing any useful action.

In response, researchers have devised novel mechanisms for recovering execution in the presence of errors [13]. ASSURE [26], in particular, presents a powerful system that enables applications to automatically self-heal. Its operation revolves around the understanding that programs usually include code for handling certain anticipated errors, and it introduces the concept of rescue points (RPs), which are locations of error handling code that can be reused to gracefully recover from unexpected errors. In ASSURE, RPs are the product of offline analysis that is triggered when a new and unknown error occurs, but they can also be the result of manual analysis. For example, RPs can by identified by examining the memory dump produced when a program abnormally terminates. Also, they serve a dual role, first they are the point where execution can be rolled back after an error occurs, and second they are responsible for returning a valid and meaningful error to the application (*i.e.,* one that will allow it to resume normal operation).

Regrettably, deploying RPs using ASSURE is not straightforward, but it demands that various complex systems are present. For instance, to support execution rollback, applications are placed inside the Zap [19,15] virtual execution environment, while RP code is injected using Dyninst [4]. Zap is a considerably complex component that is tightly coupled with the Linux kernel, and requires maintenance along with the operating system (OS). In practice, RPs are a useful but temporary solution for running critical software until a proper solution, in the form of a dynamic patch or update, is available. It is our opinion that RPs have not been widely used mainly because of the numerous requirements, in terms of additional software and setup, of previous solutions like ASSURE.

We propose REASSURE, a self-contained mechanism for healing software using RPs. REASSURE assumes that a RP has already been identified, and needs to be deployed quickly and in a straightforward manner. It builds on Intel's PIN dynamic binary instrumentation (DBI) framework to install the RP and provide the virtual execution environment for rolling back execution. As Pin itself is simply an application, installation is simple and very little maintenance (or none at all) is necessary. Furthermore, REASSURE does not need to be continuously operating or even present, but it can be easily installed and attached only when needed. Disabling it and removing it from a system is equally uncomplicated, since it can be detached from a running application without interrupting its operation. Combined with a dynamic patching mechanism [4,9,17], applications protected with REASSURE can be run and eventually patched without any interruption.

We have implemented REASSURE as a Pin tool for Linux[1]. Our evaluation with popular servers, like Apache and MySQL, that suffer from well known vulnerabilities shows that REASSURE successfully prevents the protected applications from terminating. When no faults occur, the performance overhead

[1] Interested readers can contact the authors for a copy.

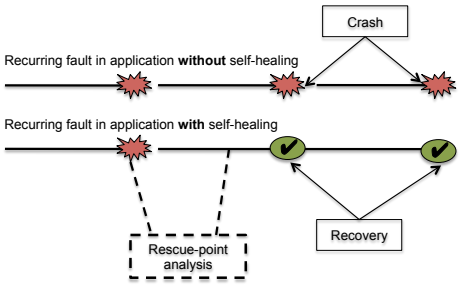


Fig. 1. Software self-healing overview. A faulty application will crash and need to be restarted every time a fault occurs. With self-healing, an analysis of the fault when it first occurs, results in the definition of a *rescue point* for the application, which allows it to gracefully recover from future occurrences of the same fault.

imposed by REASSURE varies between 1% and 115% depending on the application, while in the presence of errors there is little effect on the protected application until the frequency of faults surpasses five faults per second. We should also note that Pin supports multiple platforms (*e.g.,* Windows and Mac OS), and REASSURE can be extended to support them with little effort.

This paper is organized as follows: Section 2 presents an overview of software healing using RPs. We describe REASSURE in Sect. 3, and evaluate its effectiveness and performance in Sect. 4. Section 5 discusses limitations and future work, while related work is discussed in Sect. 6. We conclude in Sect. 7.

2 Software Self-healing Using Rescue Points

Software self-healing using RPs was first proposed in ASSURE [26], where the authors describe an architecture that enables unmodified applications to automatically heal themselves in the presence of unanticipated faults. An overview of the idea behind this scheme is presented in Fig. 1. The architecture can be decomposed into two parts. The first, is responsible for generating a RP when an unexpected error occurs, while the second is in charge of applying the produced RP on the application and recovering from future errors.

2.1 What Is a Rescue Point?

We define a rescue point as a function, preceding and encapsulating code suffering from an fault (*i.e.,* the fault it aims to mend) that contains error handling code, which can be reused to gracefully handle the unexpected error. For instance, consider the function shown in Fig. 2. It calls three other functions, namely *f1()*, *f2()*, and *f3()*. Let's assume that *f3()* contains a bug, which if triggered will terminate the application. We observe that *f3()* does not return any value, which means that it either always succeeds or simply does not handle certain conditions, such as the one causing the fault. On the other hand, the

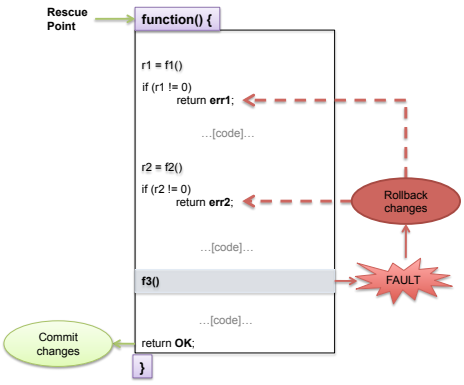


Fig. 2. Rescue point example. The function shown contains error handling code which can be used to handle errors occurring in the faulty *f3()* function.

function encompassing it contains code that handles erroneous conditions, like *f1()* and *f2()* returning an error. Therefore, we can use this function as a RP that will enable the application to self-heal from an error in *f3()*.

2.2 Rescue Point Discovery

ASSURE described a mechanism to automatically discover possible RPs and select the best fit to deploy in terms of survivability after an error occurs. Briefly, the procedure starts by profiling the application before it is deployed to discover all possible RPs. This is achieved by monitoring the values returned by the application's functions, as it is provided with fuzzed and faulty inputs. Later, when it is deployed and running normally, ASSURE takes periodic checkpoints of the application state and maintains an execution log that includes network traffic by running the application within Zap.

Concurrently, it monitors the application to detect failures and misbehavior. The simplest way to achieve this is to intercept signals such as a segmentation fault that indicates improper memory handling. Other approaches that detect memory errors can also be employed [18,8,1,21]. When an error is detected ASSURE initiates offline rescue point analysis (see Fig. 1) in a replica, which returns the application to the last checkpoint before the fault and attempts to reproduce the fault by replaying the execution log. The aim of this analysis is to detect the location of the error, thus enabling the selection of an appropriate RP. Interested readers are referred to [26] for detailed information.

Alternatively, RPs can be discovered manually. For instance, an application terminating due to a segmentation fault can be configured to dump core, a file that describes the state of the application at the time of the fault. Processing the dumped core can reveal the function containing the fault, which can be frequently used as a RP itself or assist the user to find a nearby RP fit to handle the error.

2.3 Rescue Point Deployment

In ASSURE, RPs are deployed using two systems. First, Dyninst [4] is employed to inject special code in the beginning of the corresponding function that checkpoints the application, and in case of an error returns a valid error code. Second, the Zap-based virtual environment is used to actually perform the checkpoint, as well as rollback the application when an error occurs. In the latter case, execution returns in the RP, which returns an error.

Using Zap enabled ASSURE to keep overhead low and achieve fast recovery times. Unfortunately, deploying RPs in this fashion is not very practical. Zap requires extensive modifications to the OS and cannot be dynamically installed and removed. Software self-healing targets systems that require temporary protection against known bugs until an official patch is available that properly addresses the error. As such, users are reluctant to install and maintain the additional software required to deploy ASSURE.

We offer an attractive alternative that simplifies RP deployment in the form of a self-contained mechanism built using Intel's Pin dynamic instrumentation framework. Our tool, REASSURE, only requires the Pin framework which operates on stock software and hardware. It can be dynamically applied for as long as it is required. For example, until the application is updated, or until an ingress filtering mechanism is used to block the inputs causing the fault. Afterward, it can detach itself from the application and be removed from the system.

3 REASSURE Implementation

3.1 The Pin DBI Framework

Pin [16] enables the development of tools that can augment, modify, or simply monitor a binary's execution at the instruction level. It provides a rich API that can be used by developers of tools (Pintools) to install callbacks to inspect a program's instructions and routines, as well as intercept system calls and signals. In Pin's terms, it allows the *instrumentation* of the application. Additionally, instrumentation routines can modify original code by removing instructions or by more frequently adding new code, referred to as *analysis* code. The instrumented application executes on top of Pin's virtual machine (VM) runtime, which essentially consists of a just-in-time (JIT) compiler that combines the original and analysis instructions, and places the produced code blocks into a code cache, where the application executes from.

The same block of application code can be instrumented in different ways through *versioning*. Every application thread initially executes in version zero, which corresponds to the default code cache. Instrumentation code can change the version of a running thread by adding analysis code that will change the version of the thread executing a particular instruction or block of code. When a thread switches to a new version, execution continues from the code cache of that version. If a block of code has not been instrumented for a certain version,

the instrumentation routine is called again and can install different analysis code based on the version.

Pin is actively developed and supports multiple hardware architectures and OSs. Pintools can be applied on any supported binary by either launching the binary through Pin or by attaching on an already running binary. The latter behavior is highly desirable for REASSURE, as it allows us to deploy RPs without interrupting an already executing application. We implemented REASSURE as a Pintool on Linux, but it is by no means limited to the Linux OS.

3.2 Installing Rescue Points

RPs can be installed on any callable application function. Such a function can be identified by its name or its address. The latter can be useful in cases where a binary has been entirely stripped of symbol information, and as such its functions are only identifiable by their address. In systems where the targeted binary is stripped and address space layout randomization (ASLR) [21] is used, specifying a RP's function may require additional analysis. That is because the function cannot be located by name, and its address may change due to the executable or library containing it being mapped to a different location because of ASLR. In such cases, the application can be launched without REASSURE, so we can first obtain the address where the object containing the RP's function was actually loaded. For instance, *libfoo.so* may be loaded at address *0xb6e7a000*. In Linux, such information can be obtained through the */proc* pseudo-file system. Additionally, we can statically determine the offset of the RP's function within the object. For example, function *foo()* may be defined at offset *0x800* in *libfoo.so*. By combining this information, we can calculate the address *foo()*, which in this example would be *0xb6e7a800*, and attach REASSURE on the process using the calculated RP address.

Assuming we have the means to identify RP functions, installing them is straightforward. If the function is defined by name, REASSURE first determines the address it resides in. This is accomplished by scanning the application and all its shared libraries as they are loaded. Concurrently, we scan each RP function we encounter to find at least one exit point (*i.e.,* a *ret* instruction) that will be used to return a valid error when a fault occurs. Finally, we install an instrumentation callback, which causes Pin to notify our tool whenever a new block of code is encountered. The instrumentation routine performs the following operations:

1. If a RP's entry point is encountered, analysis code is inserted to switch the thread that enters the RP to *checkpointing mode*. Primarily, this causes the thread entering the RP to switch to a different code cache version (discussed in Sect. 3.1) and saves the thread's CPU state. The *checkpointing* version of the instrumentation inserts analysis code that logs all the writes being performed by the application required for rolling back when an error occurs.
2. If a RP's exit point is encountered, analysis code is inserted to switch the thread returning from the function out of *checkpoint mode* and to normal

Table 1. Signals intercepted by REASSURE to identify and recover from program errors.

Signal	Description
SIGSEGV	Invalid memory reference/segmentation fault
SIGILL	Illegal instruction (*e.g.,* because of an invalid control-flow transfer)
SIGABRT	Abort signal sent by the *abort* system call
SIGFPE	Floating point exception (*e.g.,* divide by zero)

execution. Besides switching to the original code cache that does not log program writes, the analysis code also discards the log of writes (*i.e.,* commits the changes).

3.3 Memory Writes Logging

A RP's code, as well as all code called from it, is instrumented so as to log all the writes being performed. This *write log* serves the purpose of keeping track of all the modifications performed within a RP, so that it can be rolled back when an error occurs (*i.e.,* usually the same error that necessitated the introduction of the RP). This is achieved by augmenting every memory write instruction within a RP with analysis code that appends an entry in a dynamically expanding array, which holds the address being written and the value being overwritten. Because we are using Pin's instrumentation versioning, only the instructions being reached from within a RP are actually instrumented this way.

The analysis functions responsible for writes logging need to be carefully written to avoid certain erroneous conditions. For instance, consider a program performing an illegal memory write that causes a page fault within a RP. This memory write is also instrumented, so that the value being overwritten is saved in the log. Unfortunately, since the target address is invalid, the logging code executing before the actual write will cause the page fault instead. We have written these analysis routines in such a way that such a fault will not leave the writes log in a corrupted state (*e.g.,* with an erroneous number of entries).

3.4 Recovery from Faults

When terminal faults occur in Linux, the OS issues a synchronous signal, which if not handled will cause a process to terminate. For instance, an invalid memory reference will cause a *SIGSEGV* signal to be delivered by the OS. REASSURE intercepts such signals to identify errors occurring within RPs and initiate recovery. Table 1 lists all the signals intercepted by REASSURE to recover from program faults. Note that other OSs have similar mechanisms to synchronously notify applications of such errors. For example, Windows uses exceptions.

When REASSURE receives one of the signals in Table 1, we first check that the thread that received the signal is actually within a RP. If that is the case, we proceed to restore the values that have been overwritten since the entry to the RP and restore the saved CPU state. These actions effectively rollback the

CPU and memory modifications in single-threaded applications, and applications where the function the RP was applied on does not access shared data or interact with other threads. We discuss concurrency issues in multithreaded applications separately in Sect. 3.5. We proceed by updating the program counter to point to the *ret* instruction found during the RP's installation and use Pin's API to set the function's return value to the one specified by the RP as a valid error return value. In x86 architectures the return value is simply placed in the *eax* register. Recovery is completed by suppressing the delivery of the signal to the application and resuming execution from the updated program counter. In opposition, if one of these signals is received while the thread is not in a RP, we deliver it to the application for processing.

3.5 Concurrency

Restoring the CPU state and undoing memory writes is sufficient for recovering from faults in single-threaded applications, but this may not be the case in multithreaded applications. In general, threads share a common address space and, as such, updates made by one thread are immediately visible to all of them. Let's consider a multithreaded application with a buggy function that makes updates that affect multiple threads. It is possible that memory updates made by thread A within a RP are used by thread B to make further updates. Consequently, if an error occurs in thread A, the recovery process may leave residual data because of thread B having propagated the updates of thread A.

We address such concurrency issues by introducing *blocking* RPs that block other threads for their duration. REASSURE provides two modes of operation to accommodate blocking RPs. The first caters to applications that expect a very high rate of faults, while the second offers faster operation as long as the rate of faults is reasonable (evaluated in Sect. 4.3).

Always-on blocking mode operates by conditionally instrumenting every block of instructions with an analysis routine that blocks the executing thread when a certain flag, which is asserted by the blocking-RP upon entry, is set. Because this mode introduces frequent checks of the "block" flag, it incurs high overheads, but has low latency (*i.e.,* we can quickly activate/deactivate blocking) and is thus more appropriate for applications where faults occur very frequently.

On-demand blocking mode utilizes OS facilities to achieve better performance. In particular, we use signals (*i.e.,* the *SIGUSR2* signal) to asynchronously interrupt the remaining threads whenever a blocking-RP is entered. Similarly, to fault-related signals, REASSURE intercepts the delivery of *SIGUSR2* to install temporary blocks in receiving threads. Since the code that the thread was executing may have already been instrumented, we first remove the code currently executing from the code cache. After suppressing the delivery of the signal, Pin attempts to resume execution and since the block of code is no longer present in the code cache, our instrumentation routine is invoked again. This allows us to install an analysis routine that will block the thread. When a RP exits, we remove the blocking analysis code by once again removing the corresponding

Table 2. Applications and benchmarks used for the evaluation of REASSURE. All of applications contain exploitable bugs as described by their *common vulnerability and exposure* (CVE) id.

Application		Bug type		Benchmark
MySQL	v5.0.67	Input validation	CVE-2009-4019	MySQL's *test-insert* and *test-select*
Apache	v1.3.24	Memory corruption	CVE-2002-0392	Apache's *ab* utility
CoreHTTP	v0.5.3a	Stack overflow	CVE-2007-4060	Apache's *ab* utility
Samba	v3.0.21	Heap overflow	CVE-2007-2446	Linux's *dd* utility

instructions from the code cache. This method has the advantage of the application generally executing faster, since "blocking" code is *not* installed *de facto* for every block of code. On the other hand, since it relies on the OS to issue and deliver signals, it takes longer to block threads which may lead to decreased performance when a high rate of errors is observed.

4 Evaluation

We evaluated REASSURE along two axes. First, we show that it is able to correctly *heal* various applications that contain bugs that can cause them to abnormally terminate. Second, we evaluate the performance overhead imposed by REASSURE on these applications. In both cases, we employed existing benchmarks and tools to generate workloads. Table 2 lists the applications and benchmarks used during the evaluation. We conducted the experiments presented in this section on a DELL Precision T5500 workstation with dual 4-core Xeon CPUs (with HyperThreading disabled) and 24GB of RAM running Linux 2.6.

4.1 Recovery from Errors

We tested REASSURE's ability to heal software by triggering known bugs in the applications listed in Table 2, while concurrently running the corresponding benchmarks. When REASSURE is not employed, the applications terminate and the benchmarks are interrupted in all cases. In contrast, when using REASSURE to apply a RP that engulfs the function that causes the crash, the applications recover from the error and the benchmarks conclude successfully.

Table 3 shows the RPs applied on the applications. All applications except MySQL do not use multiple threads, but instead consist of either a single event-driven process or multiple processes. For this reason, we used non-blocking RPs for all applications besides MySQL. For the latter, even though its RP does not access shared data and consequently does not require blocking, we tested it with both RP types to demonstrate REASSURE's correctness.

Table 3. The rescue points applied to recover from the bugs listed in Table 2.

Application		Function name	Return value	Type
MySQL	v5.0.67	Item_func_set_user_var::update()	1	Non-blocking Blocking
Apache	v1.3.24	ap_bread()	-1	Non-blocking
CoreHTTP	v0.5.3a	HttpSprockMake()	0	Non-blocking
Samba	v3.0.21	switch_message()	-1	Non-blocking

4.2 Performance in the Absence of Errors

For each application in Table 2, we performed the corresponding benchmark, first with the application executing natively, then running under the Pin DBI framework, and last under REASSURE with the corresponding RP installed. This allows us to quantify the overhead imposed by REASSURE compared with native execution, as well as the relative overhead compared with the baseline, which in our case is Pin. In the tests described in this section, we did not inject any requests that would trigger the bugs each application suffers from, nevertheless the RPs listed in Table 3 were installed.

Figure 3 shows the results obtained after running 10 iterations of MySQL's *test-insert* and *test-select* benchmark tests over an 1Gb/s network link. The y-axis lists the various server configurations tested, which from top to bottom are: native execution, execution over Pin, REASSURE using a non-blocking RP, and REASSURE using a blocking RP both in on-demand and always-on blocking mode. The x-axis shows the average time (in seconds) needed to complete each benchmark, while the errors bars represent standard deviation. Note that the figure also includes standard deviation for *test-select*, but it is insignificant and thus not visible. We observe that the *test-insert* and *test-select* benchmarks take on average 24% and 53% more time to complete when running the server over REASSURE and no blocking RPs, while a significant part of the overhead is because of Pin (under Pin the tests take 18% and 46% more time). Using on-demand blocking has little effect on performance, while using always-on blocking increases the overhead to 42% and 115% respectively.

Figures 4(a) and 4(b) depict the results obtained after running 10 iterations of Apache's *ab* benchmark utility over an 1Gb/s network link for the Apache and corehttp web servers respectively. The y-axis displays the average throughput in requests per second as reported by *ab*, and the error bars represent standard deviation. We performed the experiments requesting files of different size from the web servers (listed in the x-axis), while we repeated each test with the corresponding server running: natively, over Pin, and with REASSURE (the RPs used are non-blocking). Corehttp is a single-process server and Apache was configured to only spawn a single process for serving requests to obtain comparable results.

In Fig. 4(a), we see that Apache performs approximately 4%-10% slower when run with REASSURE and the greater part of the overhead is because of Pin. We also notice that the overhead drops as the size of the requested file increases.

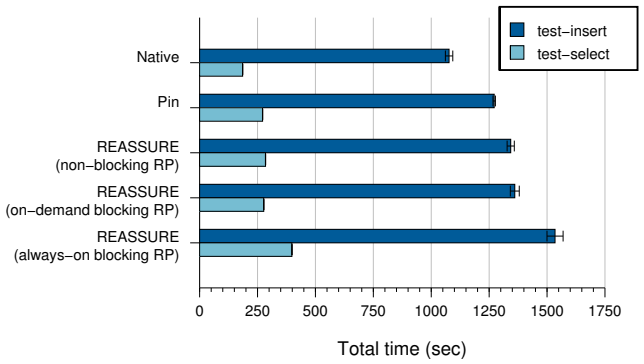


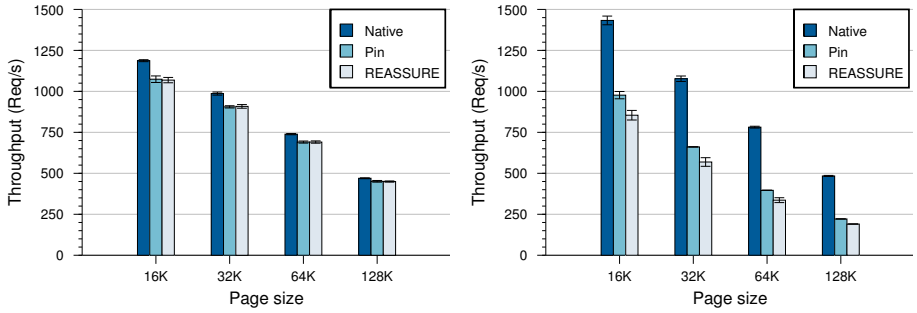
Fig. 3. MySQL performance. Time needed to complete MySQL's test-insert benchmark over an 1Gb/s network link. We apply the rescue point in three different ways: as a non-blocking RP, a blocking RP with on-demand thread blocking, and a blocking RP with always-on blocking.

This is due to the workload becoming more I/O intensive (*i.e.,* more data need to be transferred per request) and the number of requests arriving at the server shrinks. On the other hand, Fig. 4(b) shows that corehttp performs significantly worse than Apache. When running under REASSURE its throughput is reduced by approximately 40%-60%, while even when running under Pin we observe a 31%-54% reduction in throughput. There are two reasons corehttp performs so poorly. First, it is the only application where the RP is actually in the critical path of execution and it is entered for every performed request. Second, corehttp consists of many and short lived function calls that require additional processing by Pin, which be design receives control before performing any indirect control transfer like a function return. Note that the performance of code running within a RP greatly depends on parameters like the initial size of the writes log described in Sect. 3.3. If the RP is in the critical path, as in the case of corehttp, and contains many memory writes, the log will have to be frequently enlarged to accommodate the application. In the experiments described in this section, the initial size of the writes log, as well as the step used to enlarge it, is 50000 entries.

Finally, Fig. 5 shows the results of copying an 100MB file to a directory shared through samba over an 1Gb/s network link. The y-axis shows the average transfer rate (in MB/s) achieved by the *dd* utility. Once again, we performed 10 iterations of each test and we display standard deviation using error bars. We observe that when running the samba server over REASSURE there is a negligible drop in the transfer rate (approximately 1%), even though the installed RP is entered on every file transfer request.

4.3 Performance in the Presence of Errors

We complemented the experiments in the previous section by performing a set of tests against the Apache web server and the MySQL DB server running over RE-



(a) Apache (b) Corehttp

Fig. 4. Web server performance. We used Apache's *ab* benchmark utility to measure the throughput of the Apache and corehttp web servers when requesting files of different size over an 1Gb/s network link.

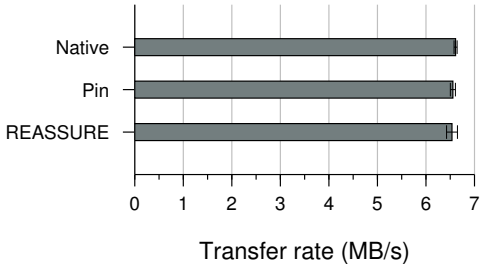


Fig. 5. Samba performance. We used the *dd* utility to copy an 100MB file containing randomly generated data to a directory shared using samba. The shared directory was mounted on a remote host over an 1Gb/s network link.

ASSURE and in the presence of errors. For Apache, we measured its throughput (in requests per second) using the *ab* utility to request a 16KB file, while concurrently we issued requests with varying frequency that triggered the server's fault, which was protected by a non-blocking RP. Figure 6 shows the results of this experiment. The x-axis is in logarithmic scale and corresponds to the time interval (in seconds) used to submit a faulty request to the server (*i.e.,* we attempted to crash the server every x seconds). When there is an one second or longer interval between the attacks to the server, it performs as well as when no errors occur, while at the same time it "heals" from the occurring errors. As the frequency of the attacks increases the attainable throughput drops. Finally, if errors occur continuously (zero seconds injection interval) the server still survives, even though throughput is greatly reduced.

In Fig. 7, we show the results obtained from running MySQL's *test-select*, while faults were injected as in the experiment described above. The y-axis

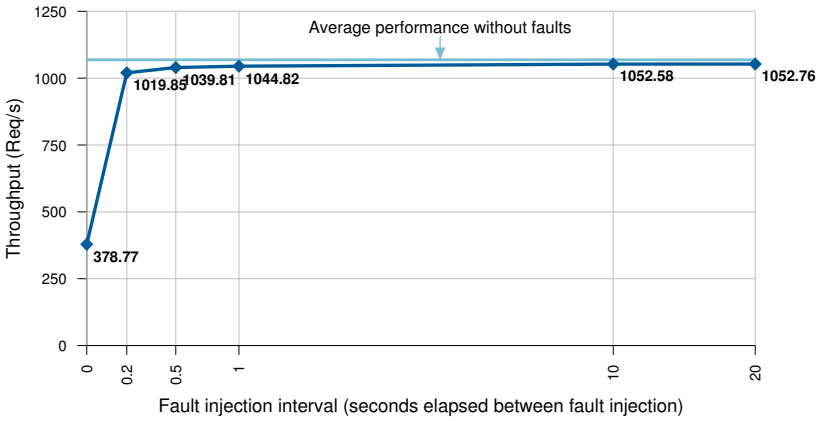


Fig. 6. Throughput of Apache web server as the number of faulty requests changes. Measured using Apache's *ab* benchmark utility to request a 16KB file containing randomly generated data. At the same time a non-blocking RP was employed to recover from the injected errors.

shows the time needed to complete each test and the x-axis corresponds to the time interval between fault injections. Both axes are in logarithmic scale. We utilized a blocking RP to recover from the faults, both in on-demand and always-on blocking mode, and in both cases we observe that if the time between faults is one second or longer, there is only a minor decrease in performance. As the frequency of the faults increases, so does the overhead in both blocking modes. Predominantly, on-demand blocking outperforms always-on blocking, but in high fault frequencies (approximately one fault per 0.1s or less) the situation is reversed. Users of REASSURE that are able to anticipate the rate of faults, can use this knowledge to select the better performing blocking mode. Alternatively, REASSURE could also monitor the frequency of faults to automatically switch from one mode to the other (discussed in Sect. 5).

5 Limitations and Future Work

There are various issues that should be considered before deploying a blocking-RP. For instance, if the RP function attempts to acquire a lock that is already held by another thread, the application may be deadlocked as REASSURE blocks all other threads. One should also be aware of certain library calls that may obtain locks (*e.g., printf()*). Therefore, blocking-RPs should be used with prudence and only when necessary, specially considering their overhead. Some of the issues with blocking-RPs can be addressed by extending REASSURE, so that RPs can be installed just on certain parts of a function (fine-grained RPs). We could then install a rescue point within the critical region of a function, after a lock has already been obtained.

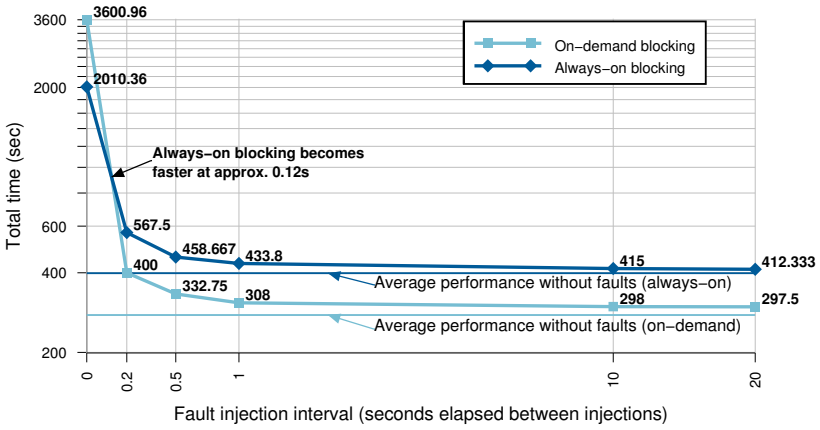


Fig. 7. Performance of MySQL DB server as the number of faulty requests changes. We measured the time needed to complete MySQL's *test-select* test in the presence of faults, and used a blocking RP to recover from the errors in always-on and on-demand blocking mode.

Some caution is also needed when setting up RPs in functions that perform certain system calls, as the current implementation of REASSURE does not rollback the effects of system calls. Frequently, this fact does not have any adverse effect on the application. For instance, retrieving the system's time, reading a random number from */dev/urandom*, and requiring more heap size do not affect the application in case of a rollback. In fact, even operations such as writing an entry in a log file or a network socket can be allowed from within a RP. While the data are still written to the destination, a rollback will leave the system in a valid state. However, this may not be the case for all applications, as RPs containing writes on critical files (*e.g.,* a DB's data file) may lead to a corrupted state. In practice, the cases where a RP should not be installed can be even more uncommon, as the error a RP is guarding against may occur before any such critical system calls, and as such the latter are inconsequential (*i.e.,* they are only executed when the error does not occur). In the future, we plan to extend REASSURE to also rollback certain system calls, like *mmap* and *munmap*. Some support for file and socket writes may also be incorporated by delaying/buffering the writes until the RP has concluded.

Finally, RPs should be applied with caution, when the targeted function updates data shared with other processes through *shared memory*. Since the updates are immediately visible to other processes, a rollback in case of error cannot guarantee that the new memory values have not been already read by another process. A possible-yet-costly solution could involve committing the updates into shadow memory, private to the thread in a RP, and upon completion copying the updates from shadow to application memory. Fine-grained RPs could also provide a workaround for applications with such issues.

In the future, we also plan to include certain optimizations that will improve performance. For instance, the write log (Sect. 3.3) is dynamically expanding and its expansion can be costly if it occurs frequently. We can "remember" the write log size required by a RP to reduce this cost. We also saw in Sect. 4.3 that depending on the frequency of errors when using blocking-RPs, it may be preferable to use always-on instead of on-demand blocking and *vice versa*. Instead, REASSURE can automatically switch between versions based on the observed fault rate.

6 Related Work

Software self-healing using RPs was first proposed in ASSURE [26]. The authors described a mechanism that can automatically analyze an application error to identify and select the appropriate RP. The deployment of the RPs was performed using a modified OS featuring the Zap [19] virtual execution environment. REASSURE does not require any modifications to the OS, and can be easily enabled and disabled. However, the RP identification component of ASSURE can be used in combination with our work.

Selective transactional emulation (STEM) [27] is a speculative recovery technique that also identifies the function where an error occurs, and it could be also used to assist in identifying RPs. Unlike REASSURE, STEM requires source code to perform the error analysis, and does not work with multiprocess and multithreaded applications. Failure-oblivious computing [25] is another speculative recovery technique that is based on the compiler inserting code to handle invalid memory writes by virtually extending the target buffer. This approach offers more robust fault response than simply crashing, but at significant performance overhead, ranging from 80% up to 500% for a variety of different applications, while it also requires recompilation of the target application.

Rebooting techniques [28,10,6] attempt to restore a system to a clean state before or after a fault. Program restart takes significantly longer time, resulting in substantial application down-time, while data loss may also occur. Micro-rebooting can be faster by only restarting parts of the system, but requires a complete rewrite of applications to compartmentalize failures. None of these techniques effectively deal with deterministic bugs, since these may recur post-restart.

Checkpoint-restart techniques [3,14] can be used in a similar fashion to rebooting, but achieve better restart times since the application can start from a checkpoint. While down time is reduced, compared with rebooting, these techniques still do not handle deterministic bugs, or bugs maliciously triggered by an attacker (*e.g.,* a DoS attack). Checkpoint-restart has been also combined with running N-versions of a program [3]. This method assumes that failures occur independently in the various versions, but introduces prohibitive costs for most applications, as multiple versions need to be maintained and run concurrently.

Automatically generating and applying patches has also been proposed, as a way to heal software [23,17,29]. Unfortunately, automatically applying patches

has not been generally adopted, due to the possibility that additional errors are introduced during the patching, or that the patched application stops behaving as expected.

7 Conclusions

We presented REASSURE, a self-contained mechanism for healing software using rescue points. REASSURE is easy to use and does not require modifications to the OS, making RPs an attractive and practical solution for temporary healing software until a patch or update is made available. It enables the reuse of existing error handling code to also handle unanticipated failures, such as the ones that can lead to the abnormal termination of an application. We have tested REASSURE with various applications including the Apache and MySQL servers, and show that it successfully allows them to recover from otherwise terminal errors. In the absence of errors REASSURE incurs an overhead between 1% and 115% depending on whether a RP is encountered frequently, and whether the application is I/O or CPU bound. We also show that when errors occur frequently, REASSURE protected applications survive, even under very adverse conditions like their continuous bombardment with errors.

Acknowledgements

This work was supported by the US Air Force and the National Science Foundation through Contracts AFRL-FA8650-10-C-7024 and AFOSR-MURI-FA9550-07-1-0527, and Grant CNS-09-14845, respectively. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, the Air Force, or the NSF.

References

1. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with WIT. In: Proc. of the Symposium on Security and Privacy. pp. 263–277 (May 2008)
2. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53, 66–75 (February 2010)
3. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. In: Proc. of the 15^{th} ACM symposium on Operating systems principles (SOSP). pp. 1–11 (1995)
4. Buck, B., Hollingsworth, J.K.: An api for runtime code patching. Int. J. High Perform. Comput. Appl. 14, 317–329 (November 2000)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8^{th} OSDI. pp. 209–224 (2008)
6. Candea, G., Fox, A.: Crash-only software. In: Proc. of the 9^{th} Workshop on Hot Topics in Operating Systems (HotOS IX) (May 2003)

7. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In: Proc. of the 10th USENIX Security Symposium. pp. 191–199 (August 2001)
8. Etoh, J.: GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/
9. Hicks, M., Nettles, S.: Dynamic software updating. ACM Trans. Program. Lang. Syst. 27, 1049–1096 (November 2005)
10. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software rejuvenation: Analysis, module and applications. In: Proc. of the 25^{th} International Symposium on Fault-Tolerant Computing (FTCS). p. 381 (1995)
11. InformationWeek: Windows home server bug could lead to data loss. http://informationweek.com/news/205205974 (December 2007)
12. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: Proc. of the 10^{th} CCS. pp. 272–280 (October 2003)
13. Keromytis, A.D.: Characterizing self-healing software systems. In: Proc. of the 4^{th} MMM-ACNS (September 2007)
14. King, S.T., Dunlap, G.W., Chen, P.M.: Debugging operating systems with time-traveling virtual machines. In: Proc. of the USENIX Annual Technical Conference (2005)
15. Laadan, O., Nieh, J.: Transparent checkpoint-restart of multiple processes on commodity operating systems. In: Proc. of the 2007 USENIX ATC. pp. 323–336 (2007)
16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proc. of the 2005 PLDI. pp. 190–200 (June 2005)
17. Makris, K., Ryu, K.D.: Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In: Proc. of the 2^{nd} EuroSys. pp. 327–340 (March 2007)
18. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the 12^{th} NDSS (February 2005)
19. Osman, S., Subhraveti, D., Su, G., Nieh, J.: The design and implementation of Zap: a system for migrating computing environments. In: Proc. of the 5^{th} OSDI. pp. 361–376 (December 2002)
20. Ostrand, T.J., Weyuker, E.J.: The distribution of faults in a large industrial software system. In: Proc. of the 2002 ACM SIGSOFT ISSTA. pp. 55–64 (2002)
21. PaX Project: Address space layout randomization (Mar 2003), http://pageexec.virtualave.net/docs/aslr.txt
22. PCWorld: Amazon EC2 outage shows risks of cloud. http://www.pcworld.com/businesscenter/article/226199/amazon_ec2_outage_shows_risks_of_cloud.html (April 2011)
23. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.F., Zibin, Y., Ernst, M.D., Rinard, M.: Automatically patching errors in deployed software. In: Proc. of the ACM SIGOPS 22^{nd} symposium on Operating systems principles. pp. 87–102 (2009)
24. Porras, P., Saidi, H., Yegneswaran, V.: Conficker C analysis. Tech. rep., SRI International (2009)
25. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., W Beebee, J.: Enhancing server availability and security through failure-oblivious computing. In: Proc. of the 6^{th} OSDI (December 2004)

26. Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., Keromytis, A.D.: ASSURE: automatic software self-healing using rescue points. In: Proc. of the 14^{th} ASPLOS. pp. 37–48 (2009)
27. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a reactive immune system for software services. In: Proc. of the 2005 USENIX ATC (April 2005)
28. Sullivan, M., Chillarege, R.: Software defects and their impact on system availability - A study of field failures in operating systems. In: Digest of Papers., 21^{st} International Symposium on Fault Tolerant Computing (FTCS-21). pp. 2–9 (1991)
29. Susskraut, M., Fetzer, C.: Automatically finding and patching bad error handling. In: Proc. of the Sixth European Dependable Computing Conference. pp. 13–22 (2006)